VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vyhledávaní vzorů ve zdrojových kódech

Pattern Discovery in Source Code

2012 **Martin Březina**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Březina**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Vyhledávaní vzorů ve zdrojových kódech
Pattern Discovery in Source Code

Zásady pro vypracování:

Cílem práce je implementace algoritmů pro vyhledávání vzorů ve zdrojových kódech.

1. Přehled současného stavu.
2. Analyzovat možnosti vyhledávání vzorů v kolekci zdojových kódů.
3. Vytvořit nástroj pro ověření projektu.
4. Experimenty a jejich vyhodnocení.
5. Závěr.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **prof. RNDr. Václav Snášel, CSc.**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2012

Martin Březina

.....

Martin Březina

Rád bych poděkoval prof. RNDr. Václavu Snášelovi, CSc. za odbornou pomoc a konzultaci při vytváření této bakalářské práce.

Abstrakt

Návrhové vzory jsou důležitým prvkem softwarového inženýrství. Řeší problémy vznikající při návrhu programu a tím návrh zjednodušují. Tato bakalářská práce se zabývá vyhledáváním vzorů ve zdrojových kódech. Nejprve si představíme návrhové vzory a jejich typy. Následuje přehled současného stavu, tudíž nástroje, které jsou již vytvořeny. V další části se věnujeme analýze, návrhu a implementaci vlastního nástroje. Tento nástroj pracuje se zdrojovými kódy jazyka C#. Popis návrhového vzoru je nástroji zprostředkován pomocí xml souboru. Stejně jako používané značky je definována i struktura popisu. Zdrojový kód i popis vzoru jsou v programu reprezentovány vlastním objektovým modelem. Oba modely jsou porovnávány, zda model zdrojového kódu obsahuje prvky modelu návrhového vzoru. V případě úspěšného vyhledání jsou uživateli poskytnuty výsledky s odpovídajícími prvky modelu zdrojového kódu.

Klíčová slova

návrhový vzor, C#, .NET, XML, rozpoznávání vzorů, porovnávání

Abstract

Design patterns are an important element of software engineering. They solve problems arising during program designing which makes designing easier. This thesis deals with pattern discovery in source code. First, we introduce design patterns and theirs classification. Followed by summary of the current state, the tools developed before. In the next section we focus on analyze, design and implementation of this tool. The tool works with source code in C#. Design Pattern‘s description is delivered to the tool using xml file. Tags used for description are defined as well as their structure. Source code and pattern description are represented in the program by an own object model. Both models are compared whether the source code model contains elements of the design pattern model. In case of successful recognition there are provided results containing coresponding elements of the source code model.

Key Words

design pattern, C#, .NET, XML, pattern recognition, comparison

Seznam použitých symbolů a zkratek

AOL	–	Abstract Object Language
API	–	Application Programming Interface
AST	–	Abstract Syntax Tree
DPML	–	Design Pattern Markup Language
EDP	–	Elemental Design Pattern
GCC	–	GNU Compiler Collection
HTML	–	HyperText Markup Language
JavaXL	–	Java Extended Language
JSON	–	JavaScript Object Notation
PDL	–	Pattern Description Language
PINOT	–	Pattern Inference Recovery Tool
PTIDEJ	–	Pattern Trace Identification, Detection and Enhancement for Java
SGML	–	Standard Generalized Markup Language
SPQR	–	Systém for Pattern Query and Recognition
SQL	–	Structured Query Language
UML	–	Unified Modeling Language
VCS	–	Version control systems
W3C	–	World Wide Web Consortium
XMI	–	XML Metadata Interchange
XML	–	Extensible Markup Language
YAML	–	"YAML Ain't Markup Language"

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Při návrhu aplikací se programátor setkává s určitými problémy. Ať už jde o jednoduché problémy nebo o problémy s velkou složitostí, vždy je potřebuje vyřešit efektivně a aby fungovaly podle daných požadavků. Právě k tomuto účelu složí návrhové vzory.

Návrhových vzorů, obecně, je mnoho, jelikož je lze využívat v různých oborech, nejen ve vývoji aplikací, nelze je přesně spočítat. S návrhovými vzory se denně setkáváme v běžném životě a ani o nich nemusíme vědět. Ať už jde o nábytek v bytě nebo venku o domy či silnice, vše je založeno na nějakých návrhových vzorech.

V softwarové sféře se postupem času některé již dříve naprogramované aplikace stávají zastaralými a už neposkytují nějaké moderní funkce anebo je jen potřeba přidat nějakou funkci. A například, kdy kvůli zastaralosti použitých technologií není vhodné rozšiřovat danou aplikaci, je naopak vhodné navrhnout aplikaci novou. Nová aplikace může nabídnout lepší stabilitu, vzhled nebo přepracované uživatelské prostředí.

Pokud jde o jednoduchou aplikaci, není ani třeba se zaobírat jakým stylem byla napsána ta předchozí, ale pokud jde o velice rozsáhlou aplikaci, bylo by za vhodno zjistit, jakým způsobem jednotlivé části spolupracovaly. Pokud toto budeme vědět, velice nám to zjednoduší návrh celé aplikace. A právě zde chceme najít, podle jakých návrhových vzorů byla předchozí aplikace napsána.

Cíl bakalářské práce

Cílem této práce je vytvořit přehled současného stavu ve vyhledávání vzorů a následné vytvoření vlastního nástroje pro rozpoznávání návrhových vzorů ve zdrojových kódech. Dále je potřeba najít způsob jakým programu předáme popis návrhového vzoru. Tento popis by mělo být jednoduché vytvořit a měl by obsahovat všechny potřebné informace ze strany návrhového vzoru. Výstupem programu bude seznam možných kandidátů na daný vzor.

Práce je rozdělena na šest částí. První část bude pojednávat obecně o návrhových vzorech a jejich typech. Druhá část bude obsahovat přehled současného stavu, tj. přehled nástrojů, které již byly vyvinuty. Ve třetí části se budeme věnovat možnostem vyhledávání a analýze nástroje. Čtvrtá část bude obsahovat vlastní implementaci nástroje. V páté části ověříme nástroj pro vyhledání několika návrhových vzorů v testovacích datech a vyhodnotíme úspěšnost. V poslední, šesté, části zhodnotíme nástroj jako celek a jeho výsledky.

2 Návrhové vzory

Termín návrhový vzor nepochází, jak by se mohlo zdát, z počítačové oblasti, ale z architektury. Jako první jej použil Christopher Alexander s kolegy v jejich knize [1]. Určitě všichni známe nějaké architektonické slohy (baroko, gotika, atd.) a právě každý tento sloh se vyznačuje nějakými specifickými prvky, které se na různých stavbách stejného slohu opakují. Tyto stejné specifické prvky definují daný sloh, a tudíž jsou jeho návrhovým vzorem.

S návrhovými vzory obecně se setkáváme každý den. Například automobil má mimo jiné čtyři kola, motor a volant, takto se dá popsat většina aut, tudíž je to určitý návrhový vzor. Nebo obyčejný stůl, většina stolů vypadá prakticky stejně. Proč? Protože vycházejí ze stejného vzoru. Podobných příkladů je mnoho a tak se bezesporu každý s nějakým takovým návrhovým vzorem určitě setkal. Proto se není čemu divit, že se návrhové vzory objeví i v počítačové oblasti, konkrétně při návrhu programů.

2.1 Návrhové vzory v softwarovém inženýrství

Návrhový vzor představuje řešení nějakého určitého problému, kterého se využívá ve fázi návrhu programu [2]. Toto řešení musí být ověřené a řádně odzkoušené. Návrhový vzor poskytuje řešení, které opakovaně funguje, tudíž nelze za návrhový vzor označit vlastní návrh, který není ověřen při různých aplikacích. Návrhový vzor není částí kódu nebo knihovnou, která by se dala vložit do programu. Ale jedná se o popis řešení nějakého problému. Tento popis se následně aplikuje do algoritmu, ale algoritmus samotný nebude návrhovým vzorem, jen bude implementovat jeho logiku popř. chování.

V tomto oboru je tou nejzákladnější knihou týkající se návrhových vzorů Design Patterns: Elements of Reusable Object-Oriented Software [2]. Tato kniha je také označována jako kniha GoF. GoF neboli Gang of Four si říká skupina autorů této knihy. Mezi další důležité knihy týkající se návrhových vzorů patří např. Patterns of Enterprise Application Architecture od Martina Fowlera [3].

Naproti návrhovým vzorům existují i antivzory. Antivzory jsou vzory, které oproti klasickým vzorům opakovaně nefungují. To znamená, že i když se algoritmus může zdát, že by mohl fungovat správně, tak nebude.

Návrhové vzory se dělí na tři základní typy, podle toho jakého charakteru je řešený problém, a to na vytvářející, strukturální a vzory zabývající se chováním [4]. Názvy vzorů se většinou udávají v angličtině, protože přesné překlady bývají složité a nemusely by přesně vystihovat jeho podstatu.

2.1.1 Vytvářející vzory (Creational Patterns)

Tyto vzory řeší problémy související s vytvářením objektů v systému [4]. Snahou těchto návrhových vzorů je popsat postup výběru třídy nového objektu a zajištění správného počtu těchto objektů. Většinou se jedná o dynamická rozhodnutí učiněná za běhu programu.

- Abstract Factory
- Factory Method
- Builder

- Lazy Initialization
- Prototype
- Singleton

2.1.2 Strukturální vzory (Structural Patterns)

Tyto vzory představují skupinu návrhových vzorů zaměřujících se na možnosti uspořádání jednotlivých tříd nebo komponent v systému [4]. Snahou je zpřehlednit systém a využít možností strukturalizace kódu.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

2.1.3 Vzory zabývající se chováním (Behavioral Patterns)

Tyto vzory se zajímají o chování systému [4]. Mohou být založeny na třídách nebo objektech. U tříd využívají při návrhu řešení především principu dědičnosti. V druhém přístupu je řešena spolupráce mezi objekty a skupinami objektů, která zajišťuje dosažení požadovaného výsledku.

- Command
- Interpreter
- Iterator
- Mediator
- Observer
- State
- Strategy
- Template

3 Přehled současného stavu ve vyhledávání vzorů

Vzory jako takové nám zjednodušují práci při návrhu programu. Proč, ale bychom chtěli návrhové vzory zpětně vyhledat ze zdrojového kódu? Odpovědí může být modernizace starší aplikace. Pokud budeme znát vzory, jaké byly v předchozí aplikaci použity, návrh nové aplikace se nám ještě zjednoduší.

Někdo si může myslet, že žádného nástroje na toto vyhledávání není potřeba, že se stačí podívat na zdrojový kód a nějaký vzor tam uvidí. Ano, toto je samozřejmě možné, ale jen pokud se jedná o systém s řádově jednotkami až desítkami tříd. Pokud budeme chtít vyhledat vzory v rozsáhlejším systému, už se bez takového nástroje neobejdeme.

Nástrojů pro vyhledávání vzorů ve zdrojových kódech existuje celá řada. U žádného z nich se ale nedá říct, že odhalí všechny vzory na sto procent. Některé se zaměřují jen na konkrétní typy a některé vyhledávají obecně všechny, ale ne vždy se jim podaří vzor rozpoznat. Zde si představíme některé z nich:

3.1 Columbus, DPML

Tento nástroj [5] používá pro popis návrhových vzorů jazyk založený na XML nazvaný Design Pattern Markup Language – DPML. Popis vzoru obsahuje element `<DesignPattern name='…'>`. Uvnitř se nacházejí třídy a reprezentace typů daného vzoru. Element třídy obsahuje její id a jméno a uvnitř obsahuje informaci o dědičnosti, kde se pro identifikaci používá id třídy. Dále obsahuje elementy pro operace a členské proměnné. Dílčími elementy jsou upraveny veškeré potřebné informace, jako je např. návratový typ operace nebo její atributy.

Vyhledávání začíná nalezením kandidátů zdrojových tříd na vzorové třídy. To je dosaženo hledáním tříd s požadovaným počtem atributů a metod s potřebnými vlastnostmi. Kontrolují se pouze parametry zadané ve vzorovém popisu. Pokud obsahuje všechny požadované vlastnosti je uložen jako kandidát na daný vzor a může se ve stejné třídě vyhledávat jiný vzor.

V druhé fázi jsou kandidáti filtrování, tak že pokud existuje vazba mezi vzorovými třídami a mezi kandidáty taková vazba není, jsou vyloučeni. Toto se stále opakuje, dokud žádnou další třídu nelze vyloučit. V další fázi se všechny kombinace kandidátů rekurzivně testují, zda spolu obsahují všechny potřebné atributy, metody a vazby vzoru. Tyto atributy a operace se také rekurzivně propojují pro vyzkoušení všech kombinací. Zde se také kontroluje, zda obsahují potřebné návratové typy a parametry.

Po nalezení kombinací se dále testuje, jestli funkce obsahují potřebné volání delegátů, vytváření objektů nebo mají požadovanou deklaraci. Jestli poslední kontrola proběhne úspěšně, potom zdrojové třídy obsahují návrhový vzor. Tyto třídy, jejich cesty a informace o řádcích a role jaké zastupují v návrhovém vzoru, jsou zobrazeny, stejně jako požadované atributy a metody.

3.2 Znovuzískání návrhových vzorů v objektově orientovaném softwaru

Proces rozpoznávání návrhových vzorů od Antoniola, Fiutema a Cristoforettiho [6] se skládá ze čtyř kroků. Prvním je AOL reprezentace. AOL (Abstact Object Language) je jazyk založený na UML a slouží k zachycení objektově orientovaného kódu do formální nezávislosti od programovacích jazyků a nástrojů. Je to UML rozšířený o textové informace a podobá se třídnímu diagramu.

Dalším krokem je získání třídních metrik. Třídní metriky jsou převedeny na AOL AST, které obsahují počty public, private a protected atributů a metod a počet asociačních, agregačních a dědičných vztahů ve kterých je třída zapojena.

Dalším krokem je vlastní rozpoznávání vzoru. Kvůli velké složitosti vyhledání mezi třídami, kde se každá porovnává s každou, je vyhledávání vyřešeno tak, že se hledají vztahy mezi třídami. Vztahy ve smyslu asociace, agregace nebo dědění, ale i různé volání metod napříč třídami. U kandidátů, kteří odpovídají vztahům vzoru, se ověřují vlastnosti jednotlivých tříd. Nakonec jsou informace o voláních obsažené v jednotlivých metodách vyjmuty z kódu.

3.3 JavaXL, PatternsBox, PTIDEJ, PDL

Java eXtended Language (JavaXL) je rozšíření Java reflection API, které pracuje za běhu a poskytuje sebepozorovací mechanizmus [7]. JavaXL byla vyvinuta pro práci během editace zdrojového kódu, což je hlavní rozdíl od ostatních rozpoznávacích algoritmů. Využívá jazyk PDL (Pattern Description Language), podobný UML, k popsání návrhového vzoru. Z toho vytvoří objektový model a pomocí PDL vytvoří model ze zdrojového kódu a poté je porovnává.

Nástroje PTIDEJ (Pattern Trace Identification, Detection and Enhancement for Java) a PatternsBox svým způsobem využívají JavaXL. Zatímco PatternsBox poskytuje asistenci při návrhu architektury nového softwaru, PTIDEJ identifikuje použité návrhové vzory již v existujícím kódu.

3.4 PINOT

PINOT (Pattern INference recOvery Tool) je plně automatizovaný nástroj pro detekci vzorů v Javě [8]. Rozpoznává dva typy vzorů – strukturální a vzory týkající se chování. PINOT zahajuje proces rozpoznávání vložením vzoru založeným na tom, co by měl nejpravděpodobněji obsahovat, aby byl co nejefektivnější v rozpoznání tohoto vzoru. Tímto se redukuje hledací prostor pruningem nejméně pravděpodobných tříd a metod.

Úplnost nástroje pro rozpoznávání vzorů je schopnost rozpoznávat varianty implementací vzoru. Z praktických důvodů se PINOT zaměřuje na rozpoznávání společných implementačních variant používaných v praxi. PINOT analyzuje pouze instance používající boolean nebo java.lang.Object typy.

3.5 SPQR

U SPQR (System for Pattern Query and Recognition) [9] je nejprve analyzován zdrojový kód pro konkrétní syntaktické konstrukce, které odpovídají konceptům, o které se zajímá. Pomocí GCC

(GNU Compiler Collection) je schopen vytvářet AST vhodný pro takovou analýzu. Jejich první nástroj, gcctree2oml, čte tento stromový soubor a produkuje XML popis funkcí objektové struktury.

Druhý nástroj, oml2otter poté čte tento objektový XML soubor a vytváří vstupní soubor funkčních pravidel pro automatizovaný nástroj pro dokazování teorémů, kde je použit Otter od Argonne National Laboratory. Otter hledá instance návrhových vzorů dokazováním na základě těchto pravidel.

Nakonec nástroj proof2pattern analyzuje výstup důkazu Otteru a vytváří zprávu s Object XML popisem vzoru, která může být použita pro další analýzu, jako třeba pro vytvoření UML diagramu.

3.6 Nástroj pro efektivní obnovování návrhových vzorů v C++ kódu

První fáze Vokáčova nástroje [10] se skládá z extrakce částí kódu ze systému pro správu verzí (VCS). Při jednoduché analýze systému je možné tuto fázi přeskočit. Výsledkem je kolekce C++ zdrojových kódů s hlavičkami i těly tříd. Tato metoda nevytváří žádné předpoklady vzhledem k umístění tříd nebo vazeb mezi třídami a soubory, ale protože VCS obecně pracuje na úrovni souborů, je pro analýzu jednodušší, když je dodržena konvence „jedna třída, jeden soubor“.

Zdrojové soubory jsou parsovány komerčním nástrojem Undestand for C++, který parsuje C++ kód na metadata. Jeho hlavním nedostatkem je, že nedokáže pracovat se šablonami (generiky), což způsobuje problémy s návrhovými vzory založenými na kolekcích. Výstupem je soubor obsahující dva druhy dat – entity a reference. Entita je jakýkoliv pojmenovaný objekt, který není klíčovým slovem, např. třída, proměnná, typ nebo soubor. Reference je vztah mezi dvěma entitami, např. deklaruje nebo volá.

V další fázi dochází k převodu entit a referencí z vlastního formátu nástroje Undestand for C++ do SQL databáze, beze změny dat. Po převedení dat je databáze indexována. Nyní je databáze připravena pro rozpoznávání strukturálních návrhových vzorů.

Rozpoznávání je dokončeno po řadě SQL dotazů, navrženy pro hledání vložené struktury. Komplikované nebo nepravidelné struktury můžou být aplikovány řetězením několika SQL dotazů v uložené proceduře. Výsledky jsou uloženy do přechodných tabulek. Nakonec jsou výsledky seskupeny a přeneseny do statistického balíku pro další analýzy.

3.7 EDPdetector4Java

EDPdetector4Java je nástroj pro vyhledávání EDP (Elemental Design Pattern) v Javě [11]. Využívá Framework RECODER k parsování zdrojového kódu a ke zjištění a prezentaci informací, které je potřeba najít. Jsou tři požadavky z pohledu RECODERu k analýze systému Javy. Prvním je, že kód musí být kompilovatelný, nesmí obsahovat syntaktické chyby, druhým, že kód musí být uzavřený, všechny reference jsou známy a třetím požadavkem je, že RECODER podporuje pouze statickou analýzu, tudíž může číst byte kód, ale nemůže ho spustit.

Tento přístup využívá reprezentaci AST zdrojového kódu ve spojení s typy výrazů, výsledných referencí a vzájemných referencí. Pro detekci EDP je definována hierarchie Visitorů.

Každý EDP je identifikován jedním Visitorem. Informace vztahující se k detekci EDP je nyní uložena v XML souboru, který je jednodušší interpretovat a navíc zkontrolovat správnost těchto informací.

Tento přístup má dvě možné alternativy, popisování AST informacemi odhalující přítomnost EDP nebo ukládání informací souvisejících s EDP do databáze pro další dotazování, které může souviset s rozpoznáváním návrhových vzorů nebo se zjišťováním metrik pro měření kvality kódu.

3.8 Efektivní rozpoznávání vzorů v objektově orientovaných návrzích

Tento přístup [12] se nezajímá o vyhledávání návrhových vzorů v kódu jako takovém, ale ve fázi návrhu v UML modelu. Pro analýzu již existujících aplikací je potřeba zpětně získat UML nějakým běžným nástrojem k tomuto určeným.

Dále se tento UML model převede na XMI soubor, který bude vstupem nástroje pro rozpoznávání vzoru. Zde se tento model bude porovnávat s již známými vzory buď metodou ohodnocování shod, nebo postupným vyhledáváním. Výstupem je výpis instancí vzorů a shodující se parametry.

4 Analýza nástroje

Možností vyhledávání návrhových vzorů ve zdrojových kódech je mnoho. Jednotlivé přístupy se neliší jen v tom, jakým způsobem vzory vyhledávají, ale i v jakém jazyce jsou napsány anebo hlavně v jakém jazyce musí být zdrojové kódy pro vyhledávání. Některé z přístupů vyhledávání jsme si již představili. Některé z nich jsou si dosti podobné a pracují na podobném principu a jiné se naopak mohou výrazně lišit.

Jak již bylo zmíněno, vyhledávat návrhové vzory je většinou potřeba ve zdrojových kódech starších aplikací. Tyto aplikace tedy pravděpodobně budou využívat nějaký „starší“ programovací jazyk. Většina dostupných rozpoznávacích nástrojů pracuje s jazykem C++, ale jsou i nástroje, které používají C, Fortran a jiné. Dokonce, pravděpodobně díky velkému rozšíření, existují nástroje pracující se stále moderním jazykem Java.

4.1 Specifikace požadavků

Vzhledem k tomu, že nástrojů pro jazyk C++ existuje mnoho, rozhodl jsem se, že v mém projektu budu vyhledávat vzory v C# kódu. Před tím, než vůbec budeme chtít začít cokoliv vyhledávat, musíme si specifikovat, co přesně budeme vyhledávat.

Nyní si rozebereme zdrojový kód, abychom znali jeho strukturu. Zdrojový kód nějakého programu většinou obsahuje namespace. Ten se skládá ze tříd (Class) nebo rozhraní (Interface) a popřípadě nějakých proměnných (Field).

Proměnná se skládá z typu proměnné, názvu a popř. nějakého modifikátoru. Třída může opět obsahovat nějaké proměnné, ale oproti namespace i metody (Method) nebo konstruktory (Constructor). Třída jako taková má nějaké jméno, opět nějaké modifikátory (public, private, abstract, static, atd.) a navíc může dědit po nějaké třídě nebo implementovat nějaké rozhraní.

Metoda je definována jménem, návratovým typem, modifikátory a parametry. Těla metod, kromě dalších proměnných, obsahují hlavní funkční část kódu. Konstruktory jsou prakticky shodné s metodami, až na nějaké rozdíly jako je například absence návratového typu. Takto by se ve stručnosti dala popsat struktura zdrojového kódu.

Pokud tato struktura (zdrojový kód) obsahuje návrhový vzor, je to nějaká její podmnožina, kterou je potřeba rozpoznat. Jelikož zdrojový kód jako text se špatně prochází a těžko by se v něm hledala jakákoliv reference, je potřeba si zdrojový kód převést na objektový model. Objektový model bude mít možnosti dle struktury kódu, jak jsme si ji popsali a měl byt co nejlépe odpovídat vlastnostem a vztahům zdrojového kódu.

Samotný model kódu by nic neřešil, pokud nebudeme mít způsob jak programu předat popis návrhového vzoru. Díky popisu budeme schopni vyhledat vzor v kódu. Jedním ze způsobů je sepsat strukturu vzoru nějakým dostupným programovacím jazykem do podoby, které by program rozuměl. Následně bude zapotřebí tento popis převést opět do objektového modelu.

Pro přepis návrhového vzoru do podoby srozumitelné programu bych, kvůli jeho jednoduchosti, zvolil jazyk XML. XML je obecný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C [13]. XML je soubor pravidel tvorby textových formátů, které

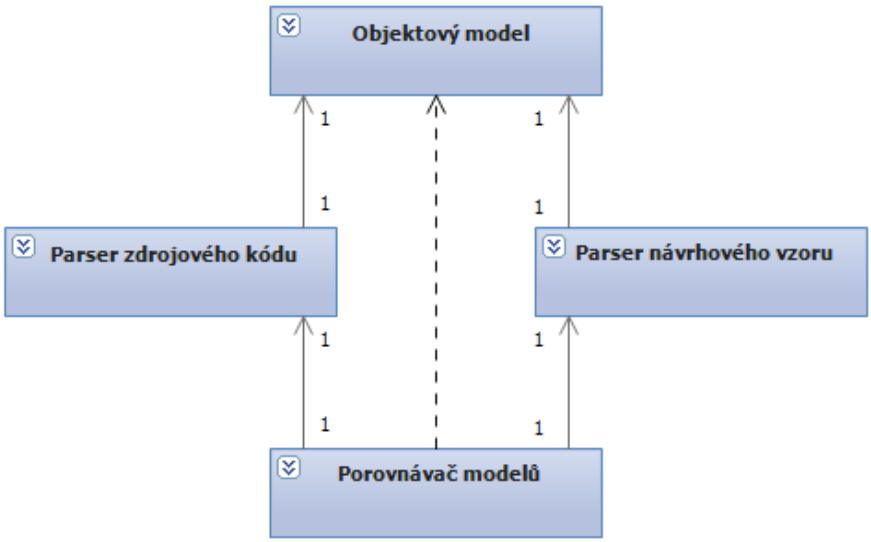
umožní vaše data uspořádat ve strukturách. XML není programovací jazyk, a k jeho zvládnutí není třeba znalostí o programování. První verze XML vyšla roku 1998 a je zjednodušenou podobou staršího jazyka SGML. Stejně jako HTML, i XML používá tzv. tagy (značky) a atributy. Zatímco však HTML přesně specifikuje, co který tag či atribut znamená a jak bude v prohlížeči zobrazen text uvnitř, XML používá tagy pouze k ohraničení částí dat, a jejich interpretace je přenechána aplikaci, která data čte. Data jsou v XML zapisovány textově, tudíž je umožněno jej otevřít běžným textovým editorem a případně je snadno editovat. Alternativou XML může být JSON nebo YAML. Zpracování XML je podporováno řadou nástrojů a programovacích jazyků. Jazyk je určen především pro výměnu dat mezi aplikacemi a pro publikování dokumentů, u kterých popisuje strukturu z hlediska věcného obsahu jednotlivých částí, nezabývá se vzhledem.

V této fázi bude potřeba oba objektové modely porovnat, ne z hlediska pojmenování tříd, ale z hlediska vlastností a vztahů mezi třídami.

Nástroj by měl splňovat následující funkční požadavky:

- Poskytovat strukturu objektového modelu.
- Vytvořit objektový model ze zdrojových kódů.
- Specifikovat XML značky pro popis vzoru.
- Vytvořit objektový model z xml popisu vzoru.
- Porovnat model kódu s modelem vzoru.
- Poskytnout výsledky porovnávání.

4.2 Konceptuální model



Obr. 4.1 Konceptuální model

Konceptuální model viz Obr. 4.1 ukazuje základní strukturu nástroje, ze kterého budeme vycházet.

V modelu je vyobrazen, jako základní entita, objektový model. Tento model bude využíván parserem zdrojového kódu, který bude číst textový zdrojový kód a převádět ho právě na tento model. Obdobně jej bude využívat i parser návrhového vzoru, který naopak bude převádět xml dokument s návrhovým vzorem.

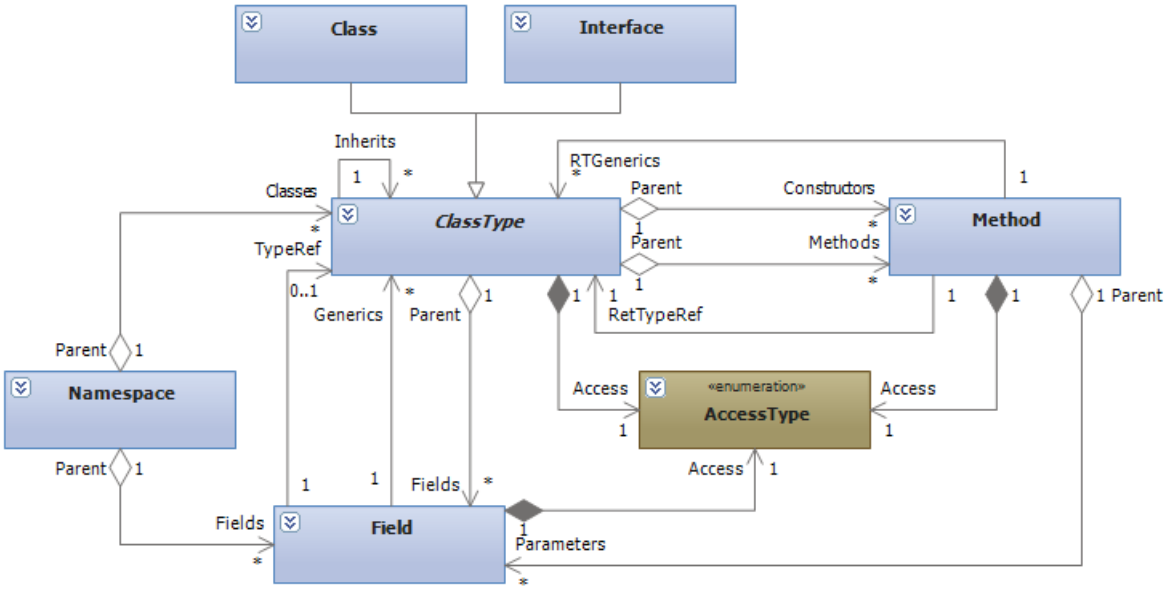
Jako poslední se zde vyskytuje porovnávač modelů. Ten má za úkol porovnat oba vytvořené objektové modely a zjistit jestli model zdrojového kódu obsahuje instanci modelu vzoru. Také bude sloužit pro výpis výsledků do souboru.

5 Návrh a implementace nástroje

5.1 Objektový model – Model

Objektový model slouží jako zjednodušená reprezentace nějaké struktury kódu, která původně není objektová. V našem případě to je zdrojový kód a popis návrhového vzoru. Tento model bude pro ostatní části dostupný jako připojitelná dll knihovna.

5.1.1 Doménový model



Obr. 5.1 Model

5.1.2 Třída Namespace

Tato třída obsahuje veškerou strukturu modelu. Namespace v případě zdrojového kódu bude obsahovat všechny třídy a položky nacházející se v jeho souborech, i v případě, kdy neobsahuje klíčově slovo namespace.

Třída je definována nepovinným jménem a obsahuje seznam tříd (ClassType), které představují třídy nebo rozhraní daného namespace. Dále obsahuje seznam proměnných (Field) a jednu metodu *GetFieldByName*, která vrací proměnnou na základě jejího jména. Dále se zde nachází slovník jmen objektů, které se nacházejí v této třídě. Tento slovník se objevuje i v dalších třídách.

5.1.3 Třída ClassType

Je to abstraktní třída reprezentující buď třídu, nebo rozhraní. Je definována svým jménem, přepínači informujících o statické nebo abstraktní třídě a typem práv přístupu AccessType. Dále obsahuje seznam třídních typů, který obsahuje třídy nebo rozhraní po kterých daná třída dědí. Dále seznamy metod, konstruktorů a proměnných.

Taktéž jako třída Namespace obsahuje metodu *GetFieldByName*. Tato metoda má stejnou funkci, ale navíc, v případě kdy proměnnou daného jména nenajde, se na ni doptá namespace, ve které se tato třída nachází. Další metodou je *ContainsPatternClassType*, která na základě základních informací (modifikátorech) o třídách rozhodne, zda tato třída může být danou vzorovou třídou.

Třída ClassType slouží jako dědičná třída pro třídu Class, reprezentující třídu, a Interface, reprezentující rozhraní.

5.1.4 Třída Method

Třída Method reprezentuje buď metodu, nebo konstruktor. Společná pro tyto dva prvky je z důvodů prakticky shodného zápisu. Metoda je definována jménem, typem práv přístupu AccessType a přepínači informujících o abstraktní, virtuální, statické nebo přepisující (override) metodě.

Dále metoda obsahuje návratový typ v textové podobě, a pokud metoda vrací nějaký datový typ definovaný tímto modelem, tak obsahuje i referenci na danou třídu. Pokud je návratový typ generický, jsou zde uloženy i reference na třídy generiky. Také je zde seznam parametrů metody, které jsou typu Field.

Tělo metody je definováno pouze seznamem volání a smyček foreach. Seznam volání je seznam trojic, kde první hodnota je proměnná, u které chceme volat metodu, druhá je právě ona metoda a třetí je seznam parametrů metody. Seznam smyček foreach, je podobný seznam. První hodnotou je proměnná, která je kolekcí, která se bude procházet, druhou je metoda, která se aplikuje na každý prvek kolekce a třetí je seznam parametrů.

Třída Method obsahuje metodu, *IsConstructor*, pro zjištění, zda je instance konstruktorem. Dále obsahuje opět stejnou metodu pro získání proměnné pomocí jména, *GetFieldByName*, která se v případě neúspěchu dotazuje dál třídy. Zde konečně najde uplatnění v budoucím vyhledávání proměnných pro volání metod a smyčky foreach. Další metodou je *ContainsPatternMethod*, která opět podle základních informací rozhoduje, zda tato metoda může být danou vzorovou metodou.

5.1.5 Třída Field

Tato třída reprezentuje nějakou položku, to je například proměnná nebo konstanta, ale v tomto případě i property (vlastnost). Definována je svým jménem, typem práv přístupu AccessType, přepínačem informujícím o statické proměnné a datovým typem.

Stejně jako u metody návratový typ, je i zde datový typ uložen textově a v případě typu definovaným modelem, obsahuje i referenci na danou třídu. Pokud je typ generický, ukládají se i reference generik.

Třída Field obsahuje jednu metodu, *ContainsPatternField*, která slouží pro porovnání základních informací a zjištění, zda tato proměnná může být danou vzorovou proměnnou.

5.1.6 Enumerátor AccessType

Tento enumerátor slouží pro nastavení práv přístupu jednotlivým objektům. Obsahuje tyto položky:

- Public

- Private
- Protected
- Internal

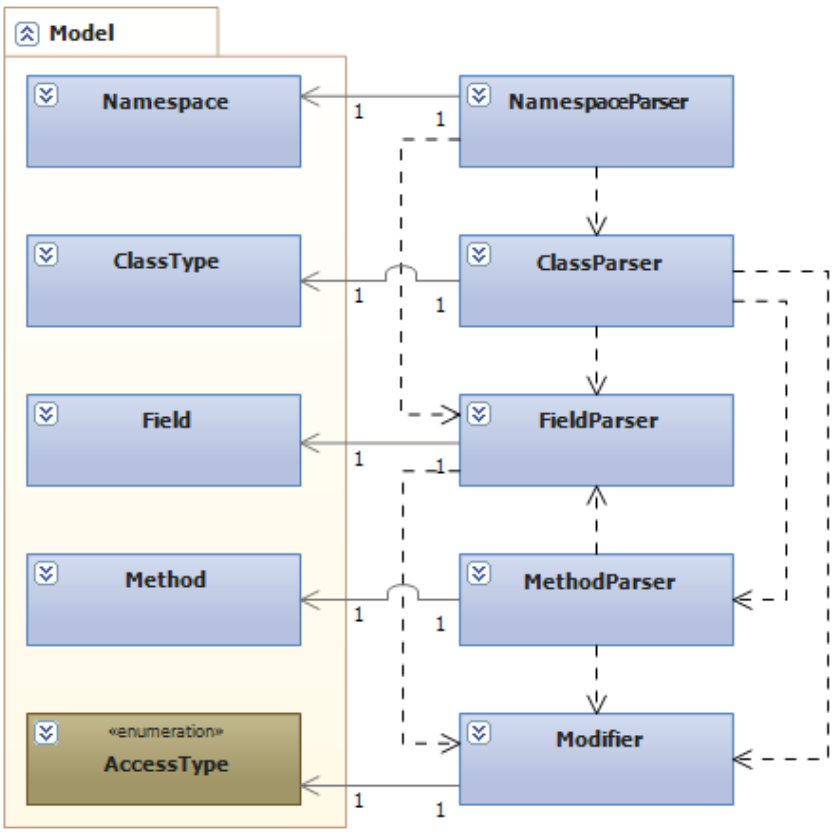
Dále je definována výjimka NoAccessTypeException, která bude sloužit při práci s přístupovým typem (AccessType).

5.2 Parser zdrojového kódu – SourceCodeParser

Tento nástroj parsuje text zdrojového kódu do objektového modelu. Obsahuje pouze statické třídy, které vstupní text převedou na požadovaný objekt. Tento parser bude dostupný jako připojitelná dll knihovna.

Vstupní zdrojové kódy nesmí obsahovat komentáře a řetězce obsahující klíčová slova zdrojového kódu. Musí být kompilovatelné, tudíž nesmí obsahovat syntaktické chyby. V opačném případě nemusí být kód pro tento nástroj čitelný. Také nedokáže pracovat s delegáty a událostmi. Zdrojový kód tyto prvky nesmí obsahovat. Dále je problematická práce s polem, konkrétně s jeho indexerem. Nástroj jej nerozpozná správně, proto je lepší využít kolekce.

5.2.1 Doménový model



Obr. 5.2 SourceCodeParser

5.2.2 Třída Modifier

Obsahuje statické metody, které slouží pro zjištění modifikátorů objektu ve vstupním textu. Vstupní text by měl obsahovat pouze ověřovaný objekt, v opačném případě může dojít k chybnému vyhodnocení.

Zjišťovanými modifikátory jsou všechny přepínače používané v modelu, tzn. modifikátor přepisu, virtuální, abstraktní a statický. Také se zde ověřuje, zda objekt obsahuje práva přístupu, pokud ano vrací se přímo AccessType typ a pokud ne, je vyvolána výjimka NoAccessTypeException.

Dále se zde nachází dva slovníky. Jeden je veřejný a slouží pro převod AccessType na odpovídající text a druhý je soukromý, který má opačnou funkci.

5.2.3 Třída FieldParser

Tato třída slouží pro vytvoření objektu Field z textu zdrojového kódu. Obsahuje tři veřejné metody.

První je *CreateField*, která na základě vstupního textu a pomocí privátních metod vytvoří nějakou určitou proměnnou. Další je *CreateParameter*, která funguje obdobně, ale vzhledem k tomu, že vytváří parametr, pracuje s jinou strukturou textu. Poslední veřejnou metodou je *IsField*, která zanalyzuje vstupní text a zjistí přítomnost proměnné. Pokud vstupní text bude obsahovat kromě proměnné i jiný text, nebude vstup považován za proměnnou.

Vzhledem k možné vysoké složitosti vlastností (property) a možnosti vytvářet vlastní proměnnou, dokáže algoritmus rozpoznat vlastnost jen jako proměnnou. Znamená to, že pokud vlastnost obsahuje kromě metod get a set i nějaký kód, který již nepracuje s vnitřní proměnnou, je tato vlastnost nesprávně rozpoznána. Tudíž tento parser dokáže pracovat pouze se základními typy vlastností.

Dále obsahuje metody pro zjištění jména a datového typu. Datový typ se zjišťuje tak, že se ignorují definovaná klíčová slova od začátku textu a jakmile se najde cizí slovo, je považováno za datový typ i s případnými generiky. Jméno proměnné se nachází na konci textu před středníkem, před závorkou nebo v případě přímého přiřazení hodnoty (reference) před znakem ‚=‘.

V případě vytváření parametru je rozlišení jména a typu mnohem jednodušší. Zde se nachází pouze tyto dvě hodnoty, případné klíčové slovo out, informující o výstupním parametru, se ignoruje.

5.2.4 Třída MethodParser

Třída MethodParser vytváří objekt Method z textu zdrojového kódu. K tomu slouží statická metoda *CreateMethod*. Dalšími veřejnými metodami jsou *IsMethod* a *IsConstructor*, které kontrolují, zda daný kód obsahuje metodu popř. konstruktor.

Dále se v této třídě nacházejí metody pro získání jména, návratového typu, parametrů nebo těla metody. Jméno se získává tak, že se najde první kulatá závorka pro parametry a slovo nacházející se před ní je jméno metody.

Návratový typ se získává tak, že se ignorují definovaná klíčová slova od začátku textu a jakmile se najde jiné slovo, považuje se za návratový typ i s případnými generiky. Pokud se po

nalezení návratového typu zjistí, že se shoduje se jménem metody, znamená to, že metoda nemá návratový typ a tudíž je konstruktorem.

Získání parametrů probíhá tak, že se najde obsah prvních kulatých závorek, který se rozdělí podle čárek a dotazováním na třídu FieldParser se parametry převedou na objekty. Tělo metody se nachází uvnitř složených závorek. Toto tělo se v případě zdrojového kódu dále neparsuje, jen se uchovává jako celek.

5.2.5 Třída ClassParser

Tato třída parsuje zdrojový kód a vytváří z něj objekt Class nebo Interface, tudíž objekt typu ClassType. Jako ostatní třídy obsahuje obdobné veřejné metody. První je *CreateClass*, která vytváří onen objekt a dotazuje se na jeho části dalších metod. Další veřejné metody *IsClass* a *IsInterface*, ověřují, zda vložený kód je třída popř. rozhraní.

Poslední veřejnou metodou je *ParseBody*. Tato metoda čte tělo třídy a dělí jej na základě konců bloků kódu (středník, uzavřená složená závorka) na části. Tyto části kódu jsou podle nalezené struktury dále parsovány na metody, konstruktory nebo proměnné.

Metoda pro nalezení jména, hledá slovo nacházející se za klíčovým slovem class nebo interface. Metoda sice rozpozná generický název, ale dále v programu se bude se jménem pracovat jako s negenerickým.

Další metodou je metoda pro získání tříd, po kterých daná třída dědí. Seznam těchto tříd se nachází za dvojtečkou za názvem třídy. Tyto třídy budou zatím obsahovat pouze název a později bude vytvořena případná reference.

5.2.6 Třída NamespaceParser

Třída NamespaceParser je základní třídou celého parsru, která řídí ostatní části a zpětně přiděluje reference objektům, které je vyžadují. Hlavní metodou je zde metoda *CreateNamespace*, která vytváří objekt Namespace. Tato metoda má jako vstupní parametr seznam řetězců, které obsahují texty zdrojových souborů. V těchto textech se najde pomocí regulárního výrazu klíčové slovo namespace a to co se dále nachází ve složených závorkách je vyjmuto a dále zpracováno. Pokud zdrojové kódy neobsahují namespace, ignorují se řádky using ze začátku textů a zbytek je považován za tělo, které je dále zpracováno.

Poté co se zpracují těla souborů, je potřeba přiřadit reference. Nejprve se přiřazují reference na dědičné třídy. Podle jména dědičné třídy uloženého při parsování určité třídy se vyhledá třída se stejným jménem a namísto objektu pouze se jménem se uloží reference. Pomocí dalších neveřejných metod se přiřadí další reference.

Třídy nesmí obsahovat atributy (např. `[Serializable]`). V opačném případě nebude blok s třídou rozpoznán jako třída a bude ignorován.

Je nezbytné, aby všechny ukládané prvky měly unikátní jména, protože veškeré reference se vyhledávají právě na základě tohoto jména.

Metoda pro parsování těla funguje na podobném principu jako stejnojmenná metoda ve třídě ClassParser. Metoda pro nastavení referencí návratových typů metod dělí návratový typ na základě

případné generiky. A každý z těchto typů se pokouší najít v seznamu získaných tříd, pokud se najde třída se stejným jménem, je na ni uložena reference. Vyhledávání v generikách funguje spolehlivě pouze na první úrovni. Tím je myšleno, že pokud jako generický typ bude uveden další generický, nemusí být správně rozpoznán.

Metoda pro nastavení datových typů proměnných pracuje na stejném principu jako metoda pro přiřazení referencí návratových typů metod, akorát pracuje nad jinými daty.

V této třídě se také nachází rozšiřující metoda *IndexOfRightBrace* pro třídu `System.String`, která v řetězci vyhledává index pravé složené závory. Vstupním parametrem je index levé závorky a vyhledává se ta pravá závorka, která levou uzavírá. Problémem vyhledávání ve zdrojovém kódu je možná přítomnost řetězců v uvozovkách nebo komentáře, které obsahují právě nějakou složenou závorku. Tímto by mohlo dojít k nepřesnému vyhledání nebo vyvolání výjimky. Proto vstupní text nesmí obsahovat tyto prvky.

5.3 Parser návrhového vzoru v XML – PatternXmlParser

Tento nástroj parsuje popis návrhového vzoru xml dokumentu do objektového modelu. Obsahuje pouze statické třídy, které převedou prvky dokumentu na požadované objekty. Tento parser bude rovněž dostupný jako připojitelná dll knihovna. Díky snadné práci s xml dokumentem je parsování mnohem jednodušší než v případě parsru zdrojového kódu. Značky používané pro popis vzoru jsou následující:

- **Pattern** – je povinná značka a obsahuje celou definici vzoru, nahrazuje namespace. Má jeden povinný atribut a tím je název vzoru – name. Může obsahovat značky Class, Interface a Field.
- **Class, Interface** – definuje třídu, rozhraní. Může obsahovat další značky Method, Constructor, Field nebo inherit. Atributy jsou:

Tabulka 5.1: Atributy značky Class/Interface

Název	Povinný	Popis
name	ano	jméno třídy/rozhraní
abstract	ne	přepínač abstraktní třídy
static	ne	přepínač statické třídy
access	ne	práva přístupu (textově prvky AccessType)

- **Method** – definuje metodu. Obsahuje značky param, call nebo foreach. Metoda může obsahovat tyto atributy:

Tabulka 5.2: Atributy značky Method

Název	Povinný	Popis
name	ano	jméno metody
ret	ano	návratový typ metody, nepovinný v případě konstruktoru
abstract	ne	přepínač abstraktní metody
virtual	ne	přepínač virtuální metody
static	ne	přepínač statické metody
override	ne	přepínač přepisující metody
access	ne	práva přístupu (textově prvky AccessType)

- **Constructor** – definuje konstruktor. Vychází z metody, tudíž má stejné prvky (Tabulka 5.2), až na nepovinný atribut návratového typu.
- **Field** – tato značka definuje proměnnou. Dále nemůže obsahovat žádné další značky, pouze tyto atributy:

Tabulka 5.3: Atributy značky Field

Název	Povinný	Popis
name	ano	jméno proměnné
type	ano	datový typ proměnné
static	ne	přepínač informující o statické proměnné (true/false)
access	ne	práva přístupu (textově prvky AccessType)

- **inherit** – definuje třídu, po které dědí třída, ve které se tento prvek nachází. Obsahuje pouze jediný povinný atribut name – jméno třídy.
- **call** – tato značka definuje volání metody dané proměnné. Dále může obsahovat pouze parametry (param) volané metody a obsahuje tyto dva atributy:

Tabulka 5.4: Atributy značky call

Název	Povinný	Popis
name	ano	jméno proměnné
method	ano	použitá metoda

- **foreach** – definuje použití smyčky foreach pro nějakou kolekci a uvnitř bloku volání metody nad prvky kolekce. Stejně jako značka call může obsahovat pouze parametry (param) volané metody a tyto dva parametry:

Tabulka 5.5: Atributy značky foreach

Název	Povinný	Popis
name	ano	jméno kolekce
method	ano	použitá metoda

- **param** – definuje parametr používaný v několika značkách. Obsahuje dva atributy:

Tabulka 5.6: Atributy značky param

Název	Povinný	Popis
name	ano	jméno parametru
type	ano	datový typ parametru; při použití s call nebo foreach nepovinný

Pokud popis bude obsahovat jiné značky než definované, budou ignorovány. Při zápisu generických datových typů je potřeba místo ostrých závorek použít jejich kódové označení (< = `<` a > = `>`).

5.3.1 Doménový model

Je shodný s doménovým modelem parsru zdrojového kódu, viz Obr. 5.2.

5.3.2 Třída Modifier

Třída obsahuje pouze dva slovníky. První převádí řetězec na hodnotu AccessType a druhý má opačnou funkci.

5.3.3 Třída FieldParser

Obsahuje jedinou metodu *CreateField*, která vytváří z xml položky Field stejnojmenný objekt.

5.3.4 Třída MethodParser

Stejně jako u SourceCodeParsru jeho veřejná metoda slouží k vytvoření Method objektu. Přitom dílčí značky param se převedou na parametry metody a poté se parsuje tzv. tělo metody. Tělo může obsahovat značky call a foreach, které se nacházejí na úrovni parametrů a převedou se na správnou strukturu a uloží do metody.

5.3.5 Třída ClassParser

Tato třída vytváří objekt typu ClassType z xml uzlu dokumentu. Funguje na podobném principu jako MethodParser, až na to, že místo parametrů vyhledává dědičné třídy pod značkou inherit. V tělu metody hledá značky pro metodu, konstruktor a proměnnou, které pomocí dílčích parsrů zpracuje.

5.3.6 Třída NamespaceParser

Třída pracuje s prvkem jménem Pattern a vytváří z něj instanci třídy Namespace. V těle hledá značky tříd, rozhraní nebo proměnných. Jakmile jsou všechny položky převedeny na objekty, probíhá přiřazování referencí. Přiřazování referencí probíhá stejně jako u SourceCodeParseru ve třídě NamespaceParser (viz 5.2.6).

5.3.7 Ukázka mapování xml popisu vzoru na ekvivalentní prvky objektového modelu

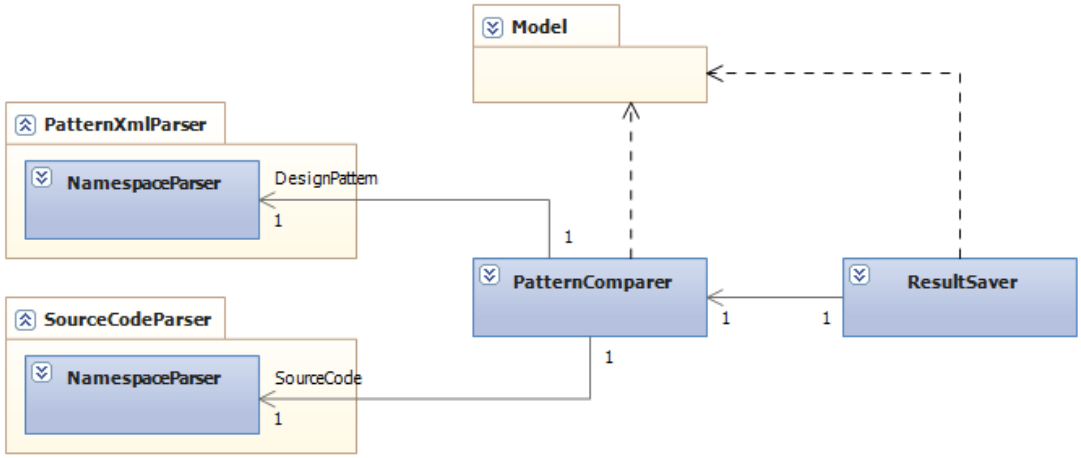
```
<?xml version="1.0" encoding="utf-8"?>

<Pattern name="Adapter Pattern">

  <Class name="Target" abstract="true">                    abstract class Target
    <Method name="Request" ret="void" abstract="true" />   {
  </Class>                                                     abstract void Request();
                                                           }

  <Class name="Adapter">                                   class Adapter : Target
    <inherit name="Target" />                              {
    <Field name="adaptee" type="Adaptee" />                  Adaptee adaptee;

    <Method name="Request" ret="void" override="true">       override void Request()
      <call name="adaptee" method="specificRequest" />       {
    </Method>                                                  adaptee.SpecificRequest();
  </Class>                                                   }
                                                           }

  <Class name="Adaptee">                                   class Adaptee
    <Method name="specificRequest" ret="void" />           {
  </Class>                                                   void SpecificRequest()
                                                             {}
</Pattern>                                                 }
```

Obr. 5.3 Ukázka mapování vzoru Adapter

5.4 Porovnávač modelů – PatternRecognizer

Hlavním úkolem tohoto nástroje je porovnání objektového modelu zdrojového kódu s objektovým modelem návrhového vzoru. Má zjistit, zda zdrojový kód obsahuje daný návrhový vzor. A to ne na základě názvů tříd, metod a proměnných, ale na základě vazeb mezi třídami, správných návratových typů metod a jejich parametrů a správných definovaných typů proměnných.

5.4.1 Doménový model



Obr. 5.4 PatternRecognizer

5.4.2 Třída PatternComparer

Tato třída obsahuje hlavní výkonnou část celého nástroje. Vlastnostmi této třídy jsou objektové modely návrhového vzoru a zdrojového kódu reprezentovány objekty typu Namespace a veřejně slouží pouze pro čtení.

Dále obsahuje adresu xml souboru návrhového vzoru a seznam adres zdrojových souborů. Při změně těchto adres automaticky dochází k aktualizaci objektových modelů a stavu porovnávání. Dalšími veřejnými vlastnostmi jsou seznamy kandidátů, které slouží jen pro čtení.

Seznamy obsahují kandidáty objektů zdrojového kódu, které odpovídají objektům návrhového vzoru. První je seznam kandidátů tříd, který obsahuje pro každou vzorovou třídu seznam tříd zdrojového kódu, které by odpovídaly této třídě. Podobné seznamy kandidátů jsou i v případě proměnných, metod a konstruktorů. U metod a konstruktorů se navíc kandidáti dělí podle tříd, ve kterých se nacházejí.

Třída obsahuje jediný konstruktor, jehož parametry jsou adresy potřebných souborů. Při nastavování adresy souboru vzoru dojde k vyvolání neveřejné metody *PreparePattern*, která pomocí knihovny XmlDocument přečte soubor a připraví ho pro další zpracování. V souboru hledá tag se jménem Pattern, pokud se jich tam bude nacházet více, ostatní budou ignorovány. V případě nastavování adres zdrojových souborů se vyvolá metoda *PrepareSource*, která přečte soubory a navíc v nich odstraní veškeré komentáře.

Po nastavení adres a přečtení souborů je potřeba tyto data parsovat na objektové modely. K tomu nám poslouží nástroje SourceCodeParser a PatternXmlParser.

Vlastní porovnávací algoritmus je rozdělen do tří úrovní. Úrovně jsou zvoleny podle míry porovnávání na namespace, třídní a úroveň metod. Na úrovni namespace porovnává metoda *NamespaceLevelCompare*, která porovnává položky, které se nacházejí přímo v namespace obou modelů. Třídní úroveň je hlubší stupeň porovnávání, kterou reprezentuje metoda *ClassLevelCompare*, která porovnává položky nacházející se uvnitř tříd a rozhraní. Nejvyšší úrovní je úroveň metod, která porovnává těla metod a je reprezentována metodou *MethodLevelCompare*.

Na úrovni namespace probíhá porovnávání tak, že se nejprve do kandidátů tříd uloží takové třídy, které úspěšně projdou základní porovnávací metodou třídy. Toto porovnávání by ale nestačilo a tak je potřeba z tohoto seznamu odebrat ty třídy, které nemají správné reference dědičných tříd. Odebírání probíhá následovně.

Pokud nějaká vzorová třída (první třída) dědí po jiné třídě (druhá třída), tak kandidátem pro první třídu, může být pouze taková třída, která dědí po třídě, která se nachází v kandidátech druhé třídy. Na podobném principu pracují i ostatní porovnávací algoritmy tohoto nástroje. Toto odebírání kandidátů probíhá opakovaně až do chvíle, kdy už dalšího nelze odebrat.

Pokud dědičná třída není referencí, tzn. obsahuje pouze jméno, znamená to, že se třída nenachází v modelu, tudíž kandidát musí dědit po této stejné třídě. V rámci této metody se ještě porovnávají proměnné. Ty musí být v potřebném počtu a správného typu i v případě generik. Následně se přidají do seznamu kandidátů na základě podobného principu jako třídy.

Dalším stupněm porovnávání je porovnávání na úrovni tříd. Metoda je závislá na kandidátech zjištěných při porovnávání na úrovni namespace, tudíž v případě potřeby je tato metoda zavolána. Zde se pomocí základního porovnávání vytvoří seznam kandidátů metod a konstruktorů. Stejně jako na nižší úrovni i zde se porovnávají proměnné a přidají se do seznamu kandidátů.

Z kandidátů metod a konstruktorů se musí vyloučit ty, které nemají správné návratové typy nebo parametry, vše se kontroluje na bázi referencí. Kandidát na metodu nebo konstruktor může obsahovat i jiné parametry než ty definované vzorem. Je to dáno tím, že metoda může být rozšířena o nějakou funkčnost, ale stále může odpovídat vzoru. Poté se odstraní z kandidátů tříd ty třídy, které nemají požadované metody, konstruktory nebo proměnné. Vše se opakuje, kvůli případnému smazání třídy, která jako kandidát byla důležitá pro referenci.

Nejvyšším stupněm porovnávání je na úrovni metod. Opět pokud nebude proveden předchozí stupeň porovnávání, bude vyvolán a v případě úspěšného porovnání se pokračuje v porovnávání. Zde se procházejí všichni kandidáti na metody a porovnávají se jejich těla. V tělech se porovnávají pouze smyčky foreach a volání metod, které byly definovány vzorovou metodou.

Tyto prvky se vyhledávají na základě regulárního výrazu vytvořeného dynamicky ze vstupních parametrů. Musí se zjistit, zda použitá metoda je správná a obsahuje potřebné parametry. Pokud metoda neodpovídá vzoru, je vyřazena z kandidátů a může dojít i k vyřazení kandidáta třídy.

Posledním blokem v porovnávání je volání metody *RemoveExtraClasses*. Tato metoda odstraňuje všechny kandidáty tříd, které by sice mohly být kandidáty dané třídy, ale vzhledem ke vztahům k ostatním třídám je nelze do vzoru zařadit. Pokud každá třída má alespoň jednoho kandidáta, tak porovnávání proběhlo úspěšně a zdrojový kód obsahuje daný návrhový vzor.

5.4.3 Třída ResultSaver

Tato třída slouží pro uložení výsledků porovnávání PatternCompareru. Je to statická třída obsahující jedinou statickou metodu *Save*. Tato metoda ukládá informace do zadaného souboru. První částí dokumentu je seznam vzorových tříd (rozhraní) a k nim odpovídající třídy zdrojového kódu. Další částí je pro každou dvojici tříd seznam vzorových metod a odpovídajících metod zdrojového kódu. Následuje stejný blok pro konstruktory. Poslední částí je seznam proměnných rozdělených podle umístění.

5.4.4 Hlavní program

Dále PatternRecognizer obsahuje třídu Program, která obsahuje metodu Main, a tudíž představuje spouštěcí část celého nástroje. Program vyžaduje přítomnost dvou textových souborů. Jedním je „DesignPatternPath.txt“, který bude obsahovat adresu vzoru, kterého budeme chtít vyhledávat. Druhým souborem je „SourceCodePaths.txt“, který na řádcích obsahuje adresy souborů zdrojových kódů. Pokud bude vzor úspěšně rozpoznán, výsledky budou uloženy v souboru „result.txt“.

6 Experimenty ve vyhledávání vzorů

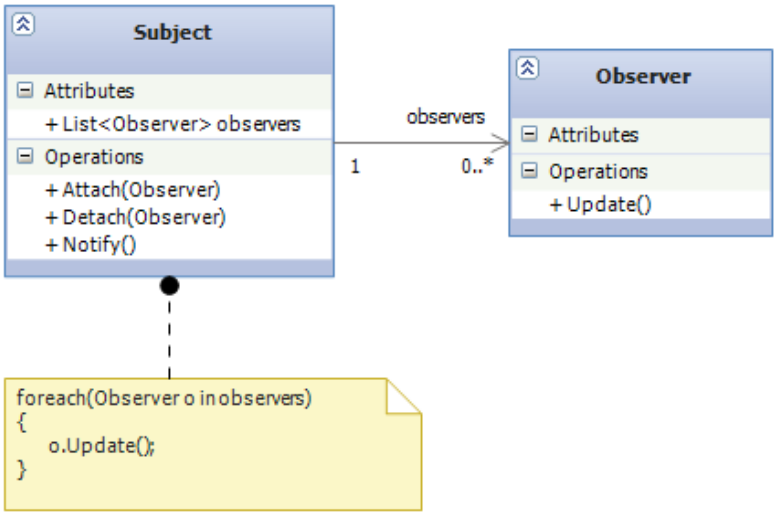
Nyní, když je nástroj pro vyhledávání vzorů ve zdrojových kódech vytvořený, je potřeba jej otestovat. Testování provedeme vyhledáváním několika vzorů v základních testovacích datech. Tímto nástroj ověříme, zda neobsahuje nějaké chyby, které případně opravíme a zjistíme skutečné možnosti nástroje ve vyhledávání.

Popisy vzorů v xml souborech jsou převedeny z popisů v diplomové práci ing. Miloše Dvořáka [4]. Testovací data pocházejí buď z mých vlastních programů anebo z internetového zdroje zabývající se mimo jiné návrhovými vzory [14].

6.1 Vzory zabývající se chováním

6.1.1 Observer Pattern

Vzor Observer (Pozorovatel) řeší problém závislosti jednoho objektu k více objektům. Závislostí je myšleno informování, nezávislým objektem, o změnách závislým objektům (pozorovatelům). Je nutné zajistit dynamickou modifikaci seznamu pozorovatelů nezávislého objektu. Snahou je oddělit nezávislý nadřízený objekt od logiky informováni závislých objektů, aby mohly být informovány bez znalosti jejich vnitřní struktury.



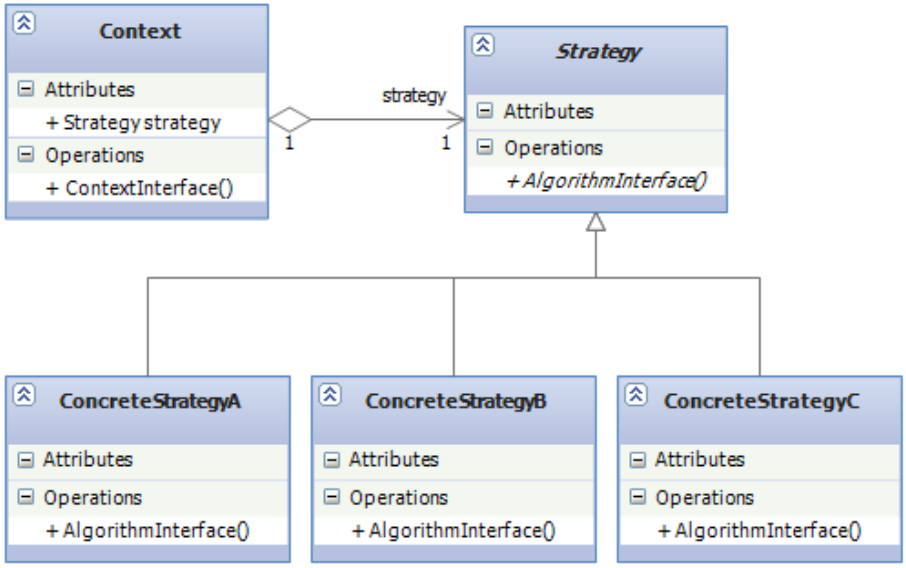
Obr. 6.1 Schéma vzoru Observer

Vzhledem k tomu, že kolekci pozorovatelů je možno zapsat více způsoby, vybral jsem tu nejpoužívanější – List. Subjekt by měl dále obsahovat nějakou zprávu, kterou si pozorovatel nějakým způsobem přečte. To už ale záleží čistě na implementaci.

Můj nástroj dokázal tento vzor úspěšně vyhledat v základních testovacích datech, tudíž se dá předpokládat, že by jej dokázal vyhledat i v mnohem rozsáhlejších datech.

6.1.2 Strategy Pattern

Strategy (Strategie) řeší problém zapouzdření různých objektů do jednoho a umožňuje tak jejich záměnu. Požadavkem tohoto vzoru je předpoklad, že existuje více podobných řešení stejného problému. Tento návrhový vzor uvažuje o skupině algoritmů, které řeší stejnou funkčnost, ale rozdílným způsobem.



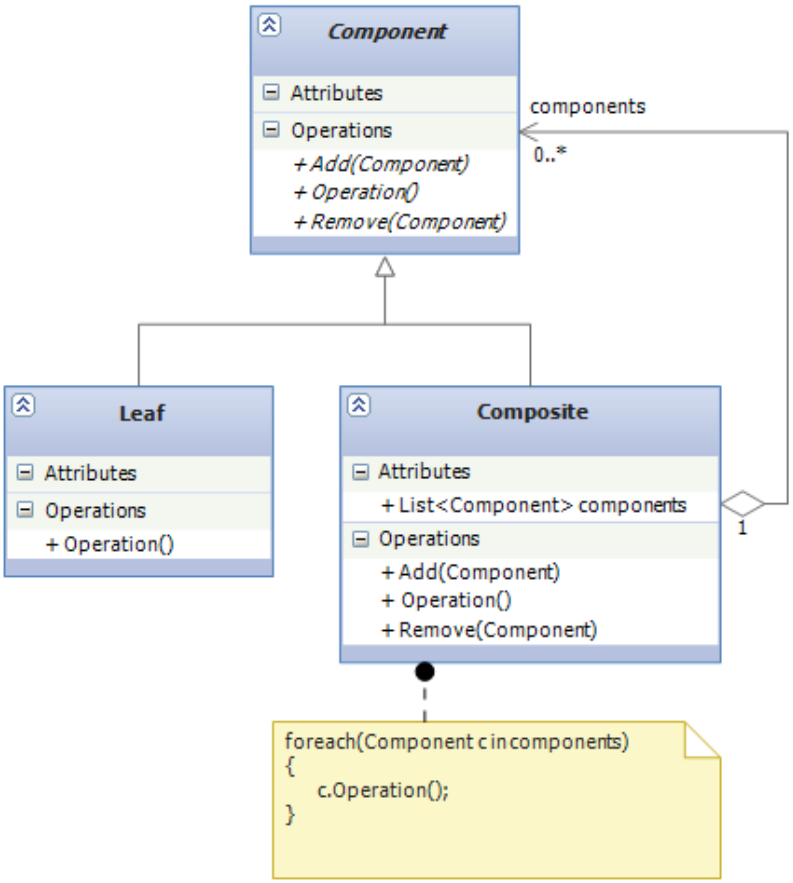
Obr. 6.2 Schéma vzoru Strategy

Vzor Strategy si zakládá na tom, aby daných ConcreteStrategy bylo více než jedna, aby měl vzor vůbec smysl. Tudíž jsem v popisu vzoru použil dva ConcreteStrategy a nástroj vyhledal vzor úspěšně ve zdrojových kódech, kde jsou definovány tři tyto třídy. Problém však nastane, když ve zdrojových kódech bude tato třída pouze jedna. Tato třída bude kandidátem na obě vzorové třídy a vzor tedy bude rozpoznán úspěšně, ačkoliv by neměl.

6.2 Strukturální vzory

6.2.1 Composite Pattern

Composite (Strom, Složený) řeší problém jak uspořádat jednoduché a složené (kompozitní) objekty, kde je k oběma potřeba přistupovat jednotným způsobem. Jednoduchý objekt dále neobsahuje referenci na jiné objekty, kdežto složený ji obsahuje. Tímto způsobem lze pomocí vzoru vytvořit stromovou strukturu. Tato obsahuje jeden kořenový uzel, který má reference na potomky tj. jednoduché objekty (listy) nebo složené objekty (větve).

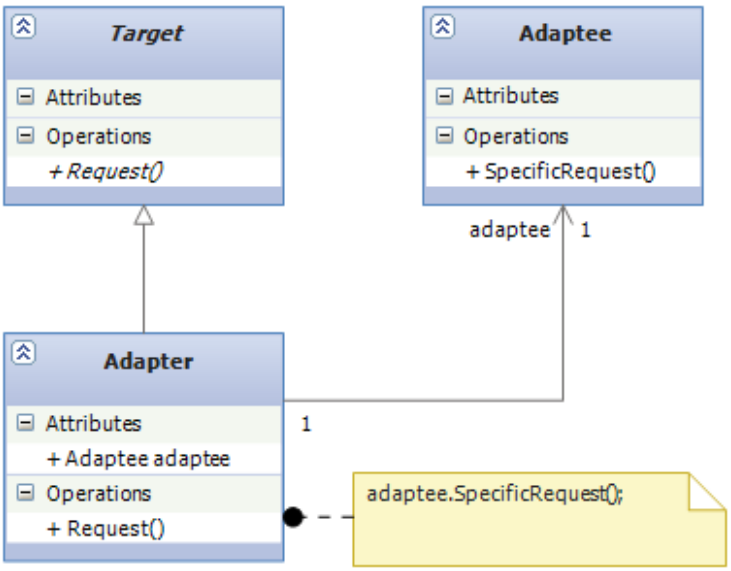


Obr. 6.3 Schéma vzoru Composite

Tento vzor byl rozpoznán téměř správně. Jelikož jednoduché objekty jsou podmnožinou složených objektů a můj nástroj nedovoluje nastavit prvky, které se nesmí objevit v dané třídě, tak všichni kandidáti na třídu Composite jsou kandidáty na Leaf. Znamená to, že rozdíl kandidátů pro Leaf a Composite, je skutečnými kandidáty pro Leaf. Ostatní prvky vzoru jsou rozpoznány správně.

6.2.2 Adapter Pattern

Adapter (Adaptér) je vzor, který řeší přizpůsobení určitě třídy, aby ji bylo možné využívat i jiným požadovaným způsobem. Problémem je zajištění konverze rozhraní jedné třídy na rozhraní jiné třídy.



Obr. 6.4 Schéma vzoru Adapter

Návrhový vzor adaptér se podařilo úspěšně vyhledat v testovacích datech. Díky tomu, že nástroj rozpoznává pouze vazby definované vzorem, byl vzor Adapter rozpoznán i v datech pro vzor Mediator a Proxy, kde se opravdu nacházely potřebné prvky.

6.3 Vytvářející vzory

Vzhledem k charakteru vytvářejících vzorů, které pracují s objekty, jejich počty a způsobu jejich šíření, je nelze vyhledávat. Způsobuje to nemožnost vytvořit popis vzoru v xml souboru a kvůli tomu také nepřizpůsobení rozpoznávacího algoritmu.

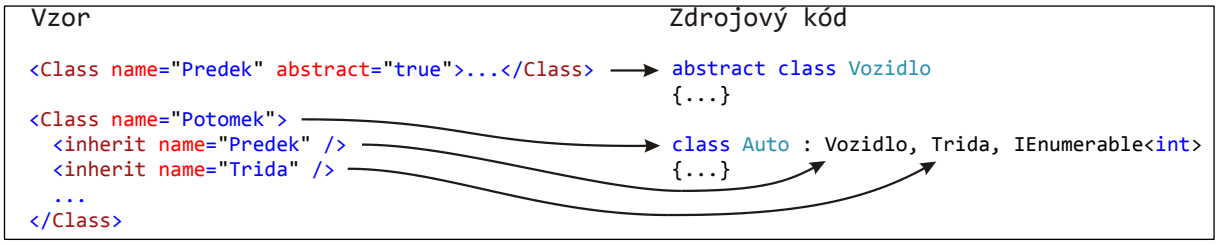
6.4 Podrobnější testování

Jednoduché testování v základních datech ale nestačí. Je potřeba nástroj otestovat i v případech kdy zdrojové kódy obsahují méně požadovaných informací anebo právě obsahují něco navíc. Tudíž potřebuje zjistit, jakým způsobem nástroj danou situaci vyhodnotí.

Nejprve budeme testovat dědičnosti (mluvíme o rozšiřování třídy a zároveň o implementaci rozhraní). V případě, kdy bude zdrojová třída dědit po stejné referenční třídě jako vzorová třída, je zřejmé, že daná třída může být kandidátem. Ale pokud bude zdrojová třída navíc dědit po jiné třídě, není zřejmé, jak toto nástroj vyhodnotí. Podobná situace může nastat, když budeme chtít vyhledat vzor s třídou, která dědí po třídě, která není definována vzorem.

První případ, kdy zdrojová třída obsahuje nějakou dědičnost navíc, vyhodnocuje nástroj tak, že porovnává pouze dědičnosti vzorové třídy. Znamená to, že pokud mezi dědičnými třídami zdrojové třídy se nacházejí správné referenční třídy dědičností vzorové třídy, tak se třída může stát jejím kandidátem a dědičné třídy navíc se ignorují. Samozřejmě pokud mezi nimi požadovaná dědičnost není, tak kandidátem nebude. U druhého případu, kdy vzorová třída dědí po třídě, která není definována vzorem, nástroj porovnává, zda zdrojová třída obsahuje dědičnost se stejným názvem jako

tato nedefinovaná třída. Pokud je taková třída obsažena, může být vzorové třídě kandidátem. Oba případy vystihuje následující obrázek, použité třídy jsou jen pro ilustraci problému (třída „Trida“ není v žádném textu definována, ilustruje nějakou obecnou třídu). Pokud ale v uvedeném zdrojovém kódu nebude Auto dědit po Trida tak třída Auto nemůže být kandidátem vzorové třídy Potomek.

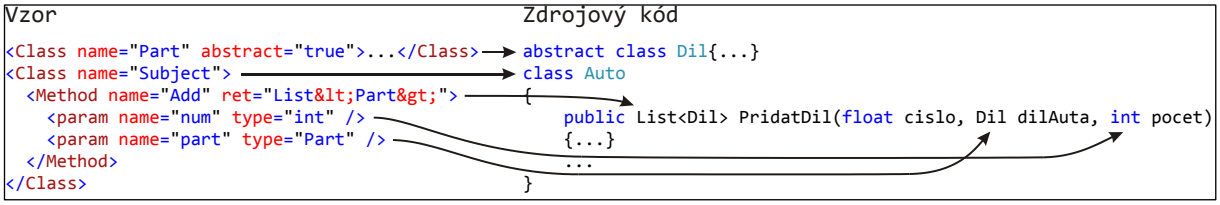


Obr. 6.5 Test dědičnosti

Co se týče metod, je potřeba otestovat parametry a návratové typy. V případě, že metoda nemá parametry, tak není potřeba nic kontrolovat (z hlediska parametrů). Když bude metoda zdrojového kódu (zdrojová metoda) obsahovat parametry stejné reference jako vzorová metoda, tak se metoda může stát kandidátem. Když ale zdrojová metoda bude obsahovat více parametrů, nebo nebudou mít konkrétní pořadí, není zřejmé, zda se metoda může stát kandidátem na vzorovou metodu. Vzhledem k datovým typům parametrů a návratového typu metody je také potřeba otestovat do jaké míry je potřeba dodržet zadání vzoru.

Řešení v případě parametrů a jejich pořadí je takové, že nástroj pořadí nevyhodnocuje, ale pouze hledá, zda mezi parametry zdrojové metody jsou parametry, jejichž datové typy odpovídají parametrům vzorové metody. Zde si také hlídá jejich požadovaný počet. V případě zvýšeného počtu parametrů zdrojové metody jsou přebytečné parametry ignorovány. Tímto je umožněno, aby zdrojové metody mohly místo vzorových instancí obsahovat i nějakou funkčnost navíc. Parametry ale nejsou ignorovány zcela. Kvůli možnosti nedodržení pořadí jsou v případě shodných datových typů uloženy mezi kandidáty na určitý parametr.

Návratový typ metody je nástrojem vyhodnocen tak, že pokud je jeho reference definována, tak se porovnává tato reference. Pokud definována není, typ musí být totožný. Pokud je návratový typ generický, porovnávají se případné reference uvnitř. Tyto vnitřní typy, ale už nesmí být dále generické, tedy jen pro případ, kdy bychom chtěli, jako generiku, zvolit definovaný typ. Pokud bychom použili nějaký nedefinovaný typ, je to možno takto použít a návratový typ bude rozpoznán správně, tudíž metoda může být kandidátem. V opačném případě nebude návratový typ správně rozpoznán. Datové typy parametrů jsou rozpoznávány obdobně. Konkrétně u následujícího obrázku, kdybychom u návratového typu metody zaměnili Dil za List<Dil>, tak nebude s ním správně pracováno, pokud bychom použili List<int> tak je vše v pořádku.



Obr. 6.6 Test parametrů metody

Uvnitř tříd se mohou nacházet volání a smyčky foreach. Zde potřebujeme otestovat použití metod a parametrů. V případě metod nás zajímá, jaké typy metod můžeme použít a u parametrů jaká je variabilita použití.

Nástroj tyto prvky vyhledává pomocí regulárních výrazů, tudíž například uvnitř smyčky foreach nedokáže odhalit, zda se místo volání nachází např. uvnitř podmínky nebo další smyčky. Tímto by mohl být výsledek ovlivněn. V obou případech v místě volání dané metody není možno, aby bylo místo správně označeno, řetězit další volání, tzn. na případně metodou vrácený prvek se musí provést další volání v odděleném příkazu. Použitými parametry nesmí být lambda výraz a také přímo hodnotové typy. Možné je použít pouze proměnné definované v parametrech nebo před metodou a ty také nesmí obsahovat nějaké doplňující informace („x+5“). Tyto parametry opět nebudou závislé na pořadí definované vzorovým prvkem a to vzhledem k variabilitě parametrů použitých metod.

Při testování proměnných se zaměřujeme pouze na jejich datový typ. U datového typu nás zajímají jeho možnosti a nutnost dodržení zadání. Vzhledem k tomu, že tento datový typ se velice podobá návratovému typu metody, je i jeho způsob vyhodnocování shodný. Tudíž i způsob přiřazování kandidátů je stejný.

Posledními prvky k otestování jsou modifikátory (abstract, static, public,…). Modifikátory jsou společné pro všechny hlavní prvky kódu a proto je na ně potřeba brát zřetel. V případě, že vzor definuje nějaký modifikátor, je zřejmé, že daný prvek zdrojového kódu by tento modifikátor měl obsahovat také, ale co v případě, že tam daný modifikátor definován není.

Pokud nějaký modifikátor definován není, tak nástroj tento prvek nekontroluje. Znamená to, že pokud vzor bude chtít nějakou metodu (nic nespecifikováno), tak kandidátem třeba může být i nějaká statická metoda. V opačném případě, když modifikátor je vzorem specifikován, je ve zdrojovém kódu striktně vyžadován. Například když vzor říká, že proměnná má být private a proměnná ve zdrojovém kódu je protected, tak nebude považována za kandidáta.

6.5 Výsledky experimentů

Tabulka 6.1: Výsledky experimentů

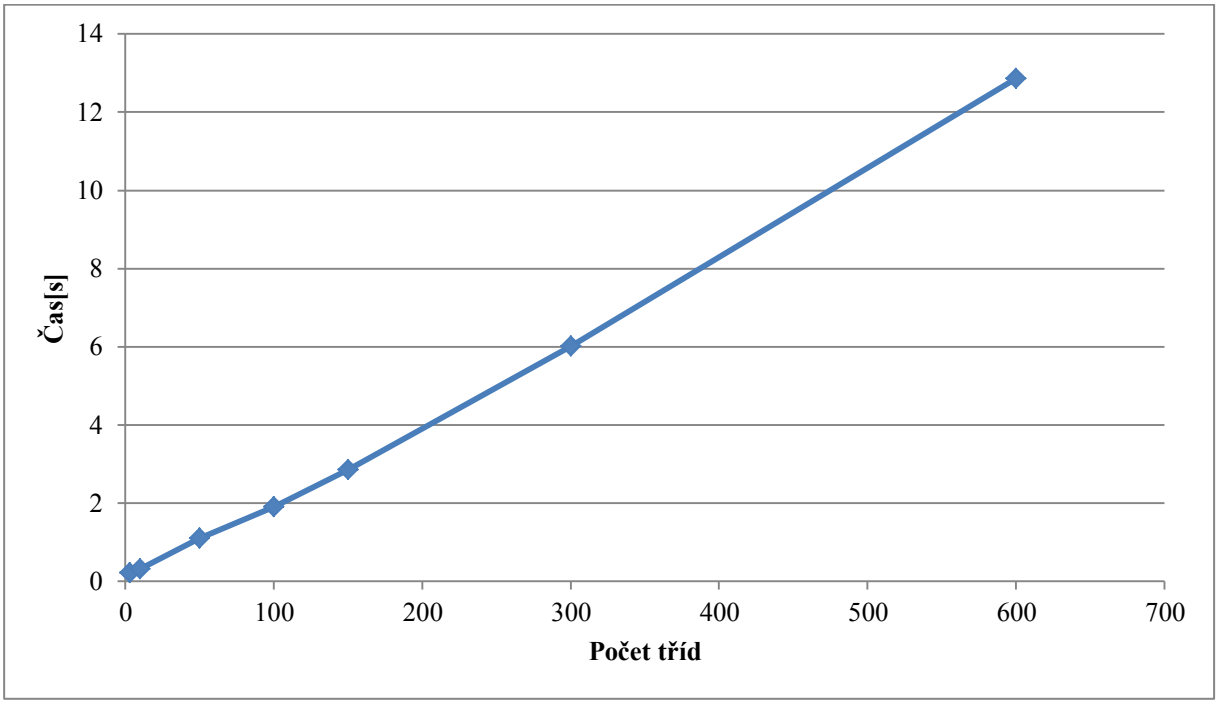
Typ vzoru	Název vzoru	Rozpoznán	Poznámka
Behavioral	Mediator	ano	
	Observer	ano	
	Strategy	ano	Problém s počtem konkrétních strategií.
Structural	Adapter	ano	
	Composite	ano	Složené objekty jsou obsaženy v jednoduchých.
	Proxy	ano	
Creational	Singleton	ne	
	Prototype	ne	

Experimenty jsme vyzkoušeli, zda je tento nástroj schopen vyhledat několik vzorů v základních testovacích datech. Z výsledků v tabulce (Tabulka 6.1) je patrné, že většina vzorů, které lze mým způsobem popsat v xml souboru, lze správně rozpoznat.

Tyto výsledky nelze přímo porovnávat s výsledky již vytvořených nástrojů. Je to z důvodu rozdílných programovacích jazyků pro vyhledávání s čímž souvisí rozdílná testovací data. Navíc má testovací data byla velmi základní, ale dá se předpokládat, že pokud by byla obsažena ve větším množství zdrojových kódů, byly by výsledky totožné.

Při testování byla odhalena jedna špatná vlastnost nástroje. A to ta, že nástroj vyhledává podmnožinu zdrojového kódu, která přesně odpovídá danému popisu vzoru. Tato vlastnost se na první pohled nemusí zdát jako špatná. Musíme si ale uvědomit, že pokud chceme v programu použít návrhový vzor, tak jej většinou musíme nějakým způsobem přizpůsobit danému problému. A právě ono přizpůsobení nemusí přesně vyhovovat našemu popisu vzoru. Z toho důvodu nemusí být vzor rozpoznán vůbec. Úspěšnost vyhledávání je tedy velice závislá na tom, jakým způsobem bude vzor popsán.

Během testování jsem také provedl test časové náročnosti vyhledávání vzhledem k počtu tříd zdrojového kódu. Zdrojové třídy byly náhodně generovány pro simulování různé složitosti reálných tříd. Měřený čas zahrnuje celý proces vyhledávání, tudíž i převedení vzoru a zdrojových kódů do objektového modelu. V tomto případě se vyhledával vzor Adapter (6.2.2), ale dá se předpokládat, že pro ostatní vzory by byly časy obdobné. Z grafu lze vidět, že s počtem tříd lineárně roste čas potřebný pro vyhledávání. V reálném případě mohou mít třídy vyšší úroveň složitosti, tudíž se výsledky mohou lišit.



Obr. 6.7 Graf časové náročnosti vzhledem k počtu tříd

Podrobným testováním jsme zjistili, jakým způsobem se chová nástroj v případě, že porovnává data, která jsou mírně odlišná. Ve většině případů se chová, tak jak bychom to normálně očekávali, tudíž by se dalo říct, že testování proběhlo úspěšně.

7 Závěr

V této bakalářské práci jsme se zabývali vyhledáváním návrhových vzorů ve zdrojových kódech. Představili jsme si základní typy návrhových vzorů a jejich funkce. Ukázali jsme si některé již vytvořené nástroje pro jejich vyhledávání.

Cílem práce bylo vytvořit nástroj pro vyhledávání vzorů ve zdrojových kódech jazyka C# a najít způsob jak nástroji předat popis návrhového vzoru. Tento popis je nástroji předáván prostřednictvím xml souboru, což je pro uživatele velice vhodné řešení z důvodu snadného vytvoření popisu a orientaci v něm. Zdrojové kódy mohou být předány jak v běžném textovém souboru, tak ve standartním C# souboru. Následně nástroj rozpoznává daný návrhový vzor ve vložených zdrojových kódech. V případě úspěšného rozpoznání, je uživateli prostřednictvím textového souboru zprostředkován výsledek, který ukazuje prvky vzoru ve zdrojových kódech.

Úspěšnost rozpoznání vzoru je závislá na jeho popisu. Pokud bude zadán přesně podle požadavků, je úspěšnost nástroje téměř stoprocentní. S jistotou odhalí, zda zadaný vzor ve zdrojových kódech je obsažen nebo není.

Nástroj ale bohužel nedokáže pracovat se všemi prvky jazyka C# a tak je vyhledávání velice omezeno. V budoucnu by bylo tedy vhodné rozšířit možnosti nástroje, aby dokázal pracovat s větším okruhem možností tohoto jazyka. Dalším prvkem nástroje, který by bylo potřeba vylepšit, je způsob vyhodnocení výsledku. Nástroj totiž sice dokáže rozpoznat, zda se v kódu daný vzor nachází a dokáže zobrazit výsledky, ale v případě neúspěšného rozpoznání by bylo zapotřebí, uživateli sdělit procentuální (ne)shodu s daným vzorem. S tímto také souvisí možné rozšíření xml popisu o nové prvky, které by popisu přidaly na univerzálnosti.

Můj nástroj je jedinečný v tom, že oproti ostatním vyhledávacím nástrojům, vzory vyhledává v jazyce C#. Při jeho programování jsem si prohloubil znalosti nejen jazyka C# ale i XML a možnosti rozšířeného porovnávaní. Nabyté zkušenosti jsem již využil při tvorbě projektů do jiných předmětů a jistě je ještě využiji.

Použitá literatura

[1] **Alexander, Christopher, Ishikawa, Sara a Silverstein, Murray.** *A Pattern Language: Towns, Buildings, Construction.* USA : Oxford University Press, 1977. 0195019199.

[2] **Gamma, Erich, a další.** *Design Patterns: Elements of Reusable Object-Oriented Software.* USA : Addison-Wesley, 1995. 0201633612.

[3] **Fowler, Martin.** *Patterns of Enterprise Application Architecture.* USA : Addison-Wesley, 2002. 978-0-321-12742-6.

[4] **Dvořák, Miloš.** Vzory. *Objekty.* [Online] 2003. [Citace: 12. 4 2012.] http://objekty.vse.cz/Objekty/Vzory.

[5] **Zsolt Balanyi, Rudolf Ferenc.** *Mining Design Patterns from C++ Source Code.* [Dokument PDF] Maďarsko : IEEE International Conference on Software Maintenance, 2003. 0-7695-1905-9.

[6] **Antoniol G., Fiutem R. a Cristoforetti L.** *Design pattern recovery in object-oriented software.* [Dokument PDF] Itálie : IEEE International Conference on Software Maintenance, 2002. 0-8186-8560-3.

[7] **Albin-Amiot H., Cointe P., Gueheneuc Y.-G. a Jussien N.** *Instantiating and detecting design patterns: putting bits and pieces together.* [Dokument PDF] Francie : IEEE International Conference on Software Maintenance, 2002. 0-7695-1426-X.

[8] **Olsson, Nija Shi a R.A.** *Reverse Engineering of Design Patterns from Java Source Code.* [Dokument PDF] USA : IEEE International Conference on Software Maintenance, 2006. 0-7695-2579-2.

[9] **Smith, J.M. a Stotts, D.** *SPQR: Flexible Automated Design Pattern Extraction.* [Dokument PDF] USA : IEEE International Conference on Automated Software Engineering, 2003. 0-7695-2035-9.

[10] **Vokáč, Marek.** *An efficient tool for recovering Design Patterns from C++ Code.* [Dokument PDF] Norsko : ETH Zurich, Chair of Software Engineering, 2006.

[11] **Arcelli, Francesca, Masiero, Stefano a Raibulet, Claudia.** *Elemental Design Patterns Recognition In Java.* [Dokument PDF] Itálie : IEEE International Workshop on Software Technology and Engineering Practice, 2005. 0-7695-2639-X.

[12] **Nguyen, Trung a Pooley, Rob.** *Effective Recognition of Patterns in Object-Oriented Designs.* [Dokument PDF] UK : Fourth International Conference on Software Engineering Advances, 2009. 978-0-7695-3777-1.

[13] **W3C.** XML Technology. *W3C.* [Online] 2010. [Citace: 13. 4 2012.] http://www.w3.org/standards/xml/.

[14] **Creative Commons.** Design Patterns. *Source Making.* [Online] [Citace: 17. 4 2012.] http://sourcemaking.com/design_patterns.

Přílohy

A. Příloha na CD

Následující tabulka popisuje adresářovou strukturu CD.

Adresář	Popis
/doc	Text bakalářské práce ve formátu PDF/A
/src/PatternRecognizer	Visual Studio 2010 Solution – Vlastní nástroj
/src/test	Testovací data
/src/test/patterns	Návrhové vzory v XML
/src/test/source_code	Základní zdrojové kódy obsahující vzory